MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Artificial Intelligence
Memo No. 130.

MEMORANDUM MAC-M- 346
April 1967

A MISCELLANEY OF CONVERT PROGRAMMING

Harold V. McIntosh,

Adolfo Guzmán.

A collection of examples to illustrate CONVERT.

A MISCELLANEY OF CONVERT PROGRAMMING

CONVERT shares with other programming languages the circumstance that it is easier to evaluate the language and to learn its use if it is possible to scrutinize a representative sample of programs which effect typical but simple and easily understood calculations. Consequently we have gathered together several examples of varying degrees of difficulty in order to show CONVERT in action. In each case the CONVERT program, written as a LISP function ready for execution in CTSS, is shown, together with the results of its application to a small variety of arguments, and a general explanation of the program, its intent, form of its arguments and method of its operation. When the notation CLOCK (()) ... CLOCK (T) appears, the time of execution has been determined, and is shown, in tenths of seconds immediately after the result has been printed.

Since there is no particular organization to the selection of examples, we here give a brief catalogue of them.

(REVERS L)	reverses the order of elements on a list
(FLIP L)	reverses a list as well as all its sublists
(CONCAT A B)	a CONVERT version of APPEND
(TIE L)	a multiple APPEND
(MERGE A B)	collates the elements of two lists
(UNMERGE L)	produces two lists, each containing alternate elements of the list L.
(NONUM E)	erases all numbers from the top level of a list
(NONUM* E)	erases the numbers from all levels of a list
(VOWELS W)	forms a list of all vowels in a list of letters or other expressions
(STAGGER L)	places the elements of L alternately at the beginning and end of a new list
(GATHER L)	inverts the operation of STAGGER, collecting together elements from the beginning and end of a list
(SUBSETS S)	produces all the subsets of the set S
(PERMUTATIONS S)	gives a list of all possible permutations of the elements of the list S
(*MARK X L)	shows the location of all X's in the list L by placing a star instead of any expression which nowhere contains an X

(FORMUL L)	transforms n-ary infix algebraic expressions to the binary prefix form
(WORD W)	makes string transformations derived from the word problem
(PERGEN S)	lists all the permutations of the set S, by first generating a skeleton which will then generate the permutations
(PB X Y)	calculates the Poisson Bracket of two algebraically defined (polynomials) X and Y
(PATHS A B M)	lists all paths between the points A and B in the network whose primitive links are given by M
(POLY)	an interactive polynomial handling program which will read an arbitrarily complicated polynomial, reduce it to an internal form which is a list of coefficients of its powers, output it in two forms, evaluate it, etc

The function (REVERS L) reverses the top level elements of its argument L; i. e., if L is (A B C D), then (REVERS L) is (D C B A).

In this example, the first argument of CONVERT is (), that is, the dictionary M is empty. In the dictionary I, second argument of the function CONVERT, we declare to X and (XXX) as variables in the UAR mode.
The third argument is L, the expression we want to reverse. Finally, the fourth argument contains one rule-set named C1, which contains a single rule:

```
( (X XXX)  ((*BEGN* (XXX)) X) )
```

The left half of this rule is the pattern (X XXX), which dissects the list we want to reverse into CAR and CDR. If this dissection is possible, that is, if (X XXX) matches, we proceed to replace the skeleton ((*BEGN* (XXX)) X), which is compossed of two pieces in this case, namely (*BEGN* (XXX)) and X.

The value of (*BEGN* (XXX)) is computed recursively, applying the entire original CONVERT program to (XXX) ---we are reversing the list (XXX) --- and then taking the contents of this result as its value.

The value of X is the expression which matched with it during the pattern comparisson, that is, the CAR of the list to be reversed.

Having computed the values of the skeletons (*BEGN* (XXX)) and X, we simply put them together, as the skeleton ((*BEGN* (XXX)) X) implies, and this is the result of our transformation.

```
DEFINE((
(REVERS (LAMBDA (L) (CONVERT
()
(QUOTE(  X (XXX)  ))
L
(QUOTE(  C1 (
( (X XXX)  ((*BEGN* (XXX)) X) )
)))
)))
))

      r convrt
      W 051.6
      load ((revers))
       NIL

      revers ( () )
       NIL

      revers ((1 2))
       (2 1)

      revers ((a b c d e f g h))
       (H G F E D C B A)

      revers (( a b c (1 2) d e ((= */)) f g))
       (G F ((= * /)) E D (1 2) C B A)

      stop
      R 1.056+7.350
```

```
print flip lisp
W 057.3

 FLIP   LISP      02/07  0057.3

   00010     DEFINE((
   00020     (FLIP (LAMBDA (L) (CONVERT
   00030     ()
   00040     (QUOTE ( X (XXX)  ))
   00050     .L
   00060     (QUOTE (  C1 (
   00070     ( (X XXX)  ((*BEGN* (XXX)) (=BEGN= X)) )
   00080     )))
   00090     )))
   00100     ))
R .700+.250
r convrt
W 057.3
load ((flip))
 NIL
flip( () )
 NIL

flip ((1 2))
 (2 1)

flip ((a b c d e f g h))
 (H G F E D C B A)

flip (( a b c (1 2) d e ((= * /)) f g))
 (G F ((/ * =)) E D (2 1) C B A)

flip (((a) (b (c d) e) f g (h i ((j k) l) m) n))
 (N (M (L (K J)) I H) G F (E (D C) B) (A))

stop
R 1.750+6.200
```

The function (FLIP L) reverses the list L, as well as all its sublists, etc., at all levels.

Compare the definition with that of (REVERS L); the rule-set C1 contains now the rule

((X XXX) ((*BEGN* (XXX)) (=BEGN= X)))

that is, the skeleton (=BEGN= X) now replaces X, and is responsable for applying the entire function to the element X, allowing in this way reversion in all levels, while the skeleton (*BEGN* (XXX)) flips the CDR.

The terminal conditions are hidden in this example; if L is an atom or an empty list, (FLIP L) is L itself, without change. This could be accomplished with an additional rule

(== =SAME=)

which is unneccessary because, as we know, when we exhaust a rule-set without finding any match for our expression, we return this last without change as a value of the transformation.

(CONCAT A B) is similar to the lisp function (APPEND A B); makes a single list of the elements of two lists.

We start by defining CONCAT as a lisp function of two variables, A and B, whose value is computed with the help of the CONVERT function. The arguments of convert are:

() -- the dictionary M is empty; no mode declarations are neccessary.

((XXX) (YYY)) -- We define in the dictionary I to (XXX) and (YYY) as fragment variables in the UAR or undefined mode.

(LIST A B) --- E, the third argument of CONVERT, is a list containing the two lists we want to tie together.

- - - - --- R, the fourth argument of CONVERT, contains one set of rules, with the name C1, with a simple rule:.

(((XXX) (YYY)) (XXX YYY))

```
print concat lisp
W 701.6

CONCAT   LISP      03/24  0701.7

DEFINE((
(CONCAT (LAMBDA (A B) (CONVERT
()
(QUOTE( (XXX) (YYY) ))
(LIST A B)
(QUOTE(C1  (
( ((XXX) (YYY))  (XXX YYY) )
)))
)))
))
R .483+.266


   r convrt
   W 702.2
load ((concat))
 NIL

concat ( () () )
 NIL

concat ((a b c ) (1 2 3 4))
 (A B C 1 2 3 4)

concat ( (a b c (d) e f) (g (h) i j k l m))
 (A B C (D) E F G (H) I J K L M)

stop
R .650+8.283

concat is similar to the lisp function APPEND.
```

(TIE L) forms a multiple APPEND of the elements of L; L is supposed to be of the form ((A1 A2 ... An) (B1 B2 ...) . . . (G1 G2 ... Gk)) and the answer or value is (A1 A2 ... An B1 B2 ... G1 G2 ... Gk)

The fourth argument of CONVERT contains a single rule, namely
(== (=ITER= (JJJ) =SAME= JJJ))

We know that (=ITER= J K L) will produce a list of the form $(L_1\ L_2\ \ldots\ L_m)$, where each element L_i is formed by replacing the skeleton L, which will probably use J inside it; the different values of L_i are due to the fact that J ranges over the elements of K.

In this case, (JJJ) ranges over the elements of =SAME=, that is, over the elements of the list we want to tie, and, for each value of (JJJ), we replace the skeleton JJJ. In this way the desired effect is achieved.

```
print tie lisp
W 154.2

   TIE   LISP      02/07  0154.2

DEFINE((
(TIE (LAMBDA (L) (CONVERT
()
(QUOTE( (XXX) (YYY) ))
L
(QUOTE(  C1 (
( == (=ITER= (JJJ) =SAME= JJJ) )
)))
)))
))
R .533+.283


r convrt
W 154.6
load ((tie))
 NIL

tie ((  (a b) (c d e f) ))
 (A B C D E F)

tie ((  (a b) (c d e f) (g h i j k) (l) (m) () (n o p) ))
 (A B C D E F G H I J K L M N O P)

tie(( () () () () () () () () ))
 NIL

tie ((  (()) (()) (()) ((())) (()) ((())) ))
 (NIL NIL NIL (NIL) NIL (NIL))

stop
R .350+7.350
```

(MERGE A B) produces, from A = (A1 A2 ... Am)
and B = (B1 B2 ... Bm) a list (A1 B1 A2 B2 ... Am Bm).

The rule set C1 comprises two rules: the first says: ((() ()) ())
that is, the merging of () with () produces ().
The second rule simply identifies X and Y --the two Car's--, puts them
X to the left of Y in the result, and begins with ((XXX) (YYY)), the two cdr's.

```
print merge lisp
W 203.9

 MERGE   LISP       02/07  0209.0

DEFINE((
(MERGE (LAMBDA (A B) (CONVERT
()
(QUOTE( X Y (XXX) (YYY) ))
(LIST A B)
(QUOTE( C1 (
( (() ()) () )
( ((X XXX) (Y YYY))  (X Y (*BEGN* ((XXX) (YYY)))) )
)))
)))
))
R .616+.433


r convrt
W 203.3
load ((merge))
 NIL

merge(  (a b c) (1 2 3) )
 (A 1 B 2 C 3)

merge(  (1 2 3 4 5 6) (a b c d e f) )
 (1 A 2 B 3 C 4 D 5 E 6 F)

merge ( () () )
 NIL

merge (  (1 2 (3) 4) (a (b) (c) d) ))"
 (1 A 2 (B) (3) (C) 4 D)

stop
R 1.216+6.433
```

UNMERGE takes a list of even length and separates its odd and even elements into two separate lists. Defined in LISP takes the form:

```
(UNMERGE (LAMBDA (L) (IF (NULL L)
                         (LIST L L)
                         ((LAMBDA (X) (LIST (CONS (CAR L) (CAR X))
                                            (CONS (CADR L) (CADR X))))
                          (UNMERGE (CDDR L))) )))
```

Again the four arguments of CONVERT are mostly trivial. However, one may see how the skeleton (=BEGN= (XXX)) corresponds to the ((LAMBDA (X) ...) (UNMERGE (CDDR L))) portion of the Lisp function, since we are dealing with a common subexpression which we wish to compute in advance. Since there is reasonable certainty of obtaining a list with two sublists, the solitary rule of the skeleton =CONT= simply serves to give these sublists names and to identify them as fragments. The variables X and Y were defined in the outer pattern, which is why =CONT= rather than =REPT= was used in proceeding to the inner rule set.

```
DEFINE((
(UNMERGE (LAMBDA (X) (CONVERT
()
(QUOTE( X Y (XXX) (UUU) (VVV) ))
X
(QUOTE(  C1 (
( () ((()()) )
( (X Y XXX)  (=CONT= (=BEGN= (XXX)) C2 (
      ( ((UUU) (VVV))  ((X UUU) (Y VVV)) )
      )))
)))
)))
))
```

```
      r convrt
      W 219.2
      load ((unmerg))
       NIL

unmerge ((a 1 b 2 c 3))
 ((A B C) (1 2 3))

unmerge ((1 a 2 b 3 c 4 d 5 e 6 f))
 ((1 2 3 4 5 6) (A B C D E F))

unmerge (())
 (NIL NIL)

unmerge ((v f e i r n y e))
 ((V E R Y) (F I N E))

unmerge ((1 a 2 (b) (3) (c) 4 d))
 ((1 2 (3) 4) (A (B) (C) D))

stop
R 1.730+7.933
```

The function (NONUM E) erases all numbers from the top level of the list E.

Basically, the rule is: ((XXX V YYY) (XXX (*REPT* (YYY)))) That is, if we identify a number in the expression, we copy the fragment to the left of this number ---that is XXX--- and repeat (that is, repeat the erasure) with YYY, the fragment to the right of this number.

The definition ov V as PAT =NUM= in the dictionary M was redundant; we could use instead the rule ((XXX =NUM= YYY) (XXX (*REPT* (YYY)))) which does not use V.

Note that we use *REPT* instead of *CONT* because, when applying the same process to (YYY), we want to go back to our original dictionary, that is, we want to unbind (XXX) and to restore it to its original definition as UAR variable. *BEGN* could be used instead of *REPT*, with identical effects. We use *REPT* instead of =REPT= since we want a fragment as value; that is to say, (=REPT= (YYY)) would produce a list without numbers in its top level, but we want the contents of such a list, so we use *REPT*.

```
DEFINE((
(NONUM (LAMBDA (E) (CONVERT
(QUOTE(
'V  PAT  =NUM=
))
(QUOTE( (XXX) (YYY) ))
E
(QUOTE ( C1 (
( (XXX V YYY) (XXX (*REPT* (YYY))) )
)))
)))
))
```

```
r convrt
W 243,0
load ((nonum))
 NIL

nonum ((a b c 1 d e f g 2 3 g ))
 (A B C D E F G G)

nonum((1 2 3 4 5 6 7 8))
 NIL

nonum (( a b 5 e 1 (6) i a n))
 (A 3 E L (6) I A N)

nonum (( (1) (2) ((3 4) 5) (6) 7 8 (3) ))
 ((L) (2) ((3 4) 5) (6) (3))



nonum (( (1) (2) ((3 4) 5) (6) 7 8 (((3))) ))
 ((1) (2) ((3 4) 5) (6) (((3))))

stop
R 1.233+3.050
```

```
nonum* erases all numbers from a given list, in all levels.
       compare with  nonum


print nonum* lisp
W 317.5

NONUM*    LISP       02/07  0317.5

DEFINE((
(NONUM* (LAMBDA (E) (CONVERT
(QUOTE(
 V  PAT  =NUM=
))
(QUOTE( (XXX) X  (YYY) ))
E
(QUOTE ( C1 (
( ((XXX) YYY)  ((=BEGN= (XXX)) (*BEGN* (YYY))) )
( (=NUM= YYY) (=BEGN= (YYY)) )
( (X XXX) (X (*BEGN* (XXX))) )
)))
)))
))
R .666+.300


r convrt
W 313.0
load ((nonum*))
 NIL

nonum* ((a b c 1 d e f g 2 3 g))
 (A B C D E F G G)

nonum* ((1 2 3 4 5 6 7 8))
 NIL

nonum* ((a b 5 e 1 (0) I a n ))
 (A B E L NIL I A N)

nonum* (( (1) (1) (2) ((3 4) 5) (6) 7 8 (9) ))
 (NIL NIL NIL (NIL) NIL NIL)

nonum* ((a (b (1 c) (2 3 89 d) (((f g 2) g)) )  ))
 (A (B (C) (D) (((F G) G))))

stop
R 2.616+7.316

observe that the declaration     V  pat  =num=
was not used and could be omitted.
```

```
VOWELS   LISP      02/07  0129.8

    00010    DEFINE((
    00020    (VOWELS (LAMBDA (W) (CONVERT
    00030    (QUOTE(
    00040    V       BUV  (=OR= A E I O U)
    00050    (VVV)   PAT  (=OR= () ((=OR= V ==) VVV))
    00060    ))
    00070    ()
    00080    W
    00090    (QUOTE ( C1 (
    00100    ( (VVV)  V)
    00110    )))
    00120    )))
    00130    ))
R .516+.266


r convrt
W 130,4
load ((vowels))
 NIL

vowels ((p r e f i x))
 (E I)

vowels ((s y n c h r ))
 NIL

v o w e l s  ((v o ?

vowels ((v o w e l s))
 (O E)

v o w e l s ? .

vowels ((a d r (i) a n a))
 (A A A)

vowels (( r o s (e) n b l u e (t) h))
 (O U E)

stop
R 1.365+3,616
```

(VOWELS W) forms a list of the vowels present in the list W, in the top level.

The dictionary M defines V in the BUV mode, and (VVV) in the PAT mode, VVV being a fragment. The pattern associated with (VVV) is (=OR= () ((=OR= V ==) VVV)), and will match with a list which is either empty or starts with (a) a vowel, or (b) anything else, and is followed by a fragment which matches VVV. That is, (VVV) tests all the elements to see if they are either V (vowels) or == (anything else); elements which match with V are stored in a list, which later becomes the value of V.

The argument R, fourth of CONVERT, contains a single set of rules, with name C1, which contains the rule ((VVV) V); that is, if the expression matches (VVV), we return V as value.

Among CONVERT programs which rearrange the elements of a list, (STAGGER L) will alternately place the elements of L at the front and end of a new list, while (GATHER L) will execute the inverse process. Thus, if L = (1 2 3 4 5 6), (STAGGER L) = (1 3 5 6 4 2), and (GATHER L) = (1 6 2 5 3 4). The key to STAGGER is the rule

```
((X Y XXX) (X (*REPT* (XXX)) Y))
```

in which successive elements X and Y are moved to the extreme ends of the bew list, and the process continued by moving the remaining elements of L, similarly transformed, into the middle of the new list. Reversal of the process depends on the rule

```
((X XXX Y) (X Y (*REPT* (XXX))))
```

in which the extreme elements are recognized, placed on the new list, followed by those elements resulting from a repetition of the process on the middle of the list.

```
gather ((0 1 2 3 4 5 6 7 8 9))
 (0 9 1 8 2 7 3 6 4 5)

stagger ((0 1 2 3 4 5 6 7 8 9))
 (0 2 4 6 8 9 7 5 3 1)

gather ((0 2 4 6 8 9 7 5 3 1))
 (0 1 2 3 4 5 6 7 8 9)
```

```
                    (STAGGER (LAMBDA (L) (CONVERT
                    (QUOTE (
                         ))
                    (QUOTE (
                         X Y (XXX)
                         ))
                    L
                    (QUOTE (*0 (
                         ((X Y XXX) (X (*REPT* (XXX)) Y))
                         )))
                         )))
```

```
                    (GATHER (LAMBDA (L) (CONVERT
                    (QUOTE (
                         ))
                    (QUOTE (
                         X Y (XXX)
                         ))
                    L
                    (QUOTE (*0 (
                         ((X XXX Y) (X Y (*REPT* (XXX))))
                         )))


                         )))
```

(SUBSETS S) produces all the subsets of the set S.

Cl is a set of two rules, the first of which says that the only subset of an empty set is an empty set.

The second rule identifies X with the CAR of S, and XXX with its CDR; we then compute (=BEGN= (XXX)), the subsets of (XXX), and we bind such result to the variable (AAA); we then replace the skeleton
(AAA (*ITER* (J) (AAA) (X J)))

This skeleton containts two parts or halves; the first is simply AAA; that is, the subsets of the Cdr are also subsets of the whole list; the other part (*ITER* (J) (AAA) (X J)) adds X to each element of the subsets of the Cdr.

```
DEFINE((
(SUBSETS (LAMBDA (S) (CONVERT
()
(QUOTE (
 X (XXX)
  ))
  S
  (QUOTE ( C1 (
  (   () (())   )
  ( (X XXX) (=SKEL= (AAA) EXPR (=BEGN= (XXX))
                    (AAA (*ITER* (J) (AAA) (X J))) ))
  )))
  )))
  ))
  r convrt
  W 648.4
  load ((subset))
   NIL

  subsets (())
   (NIL)

  subsets ((1))
   (NIL (1))

subsets ((1 2))
 (NIL (2) (1) (1 2))

subsets ((1 2 3 4))
 (NIL (4) (3) (3 4) (2) (2 4) (2 3) (2 3 4) (1) (1 4) (1 3)
 (1 3 4) (1 2) (1 2 4) (1 2 3) (1 2 3 4))

subsets ((a b c d e))
 (NIL (E) (D) (D E) (C) (C E) (C D) (C D E) (B) (B E) (B D)
 (B D E) (B C) (B C E) (B C D) (B C D E) (A) (A E) (A D) (A
 D E) (A C) (A C E) (A C D) (A C D E) (A B) (A B E) (A B D)
 (A B D E) (A B C) (A B C E) (A B C D) (A B C D E))


stop
R 2.483+9.183
```

(PERMUTATIONS S) gives a list of all possible permutations of the elements of S.

The fourth argument of CONVERT contains now two sets of rules, namely C1 and C2.

The first rule of C1 is: ((X) ((X))); if the list S has only one element X, the answer is ((X)).

(X XXX) is the pattern of the second rule, which identifies CAR and CDR; after this, (=SKEL= A EXPR X ...) defines A as the value of X --(the CAR)-- in the EXPR mode, and proceeds to replace the skeleton

(=ITER= J (=BEGN= (XXX)) (*REPT* (J ()) C2))

In this skeleton, (=BEGN= (XXX)) computes recursively the permutations of (XXX) --the CDR--, and, with J ranging over each one of the elements of this list of permutations, =ITER= proceeds to make a list of the results of replacing the skeleton (*REPT* (J ()) C2).

This last skeleton applies to (J ()) the set C2 of rules; in other words, for each element J of the permutations of (XXX), we form the subexpression (J ()) and apply to it the transformation dictated by C2. Since we call to C2 with a *REPT*, and not with a *CONT*, we, when in C2, forget all the bindings and start with our original dictionary, except that the binding or value of A is not forgotten inside C2, since it was done with an =SKEL=.

C2 transforms an expression ((J1 J2 J3 ... Jk) ()) into

((A J1 J2 ... Jk) (J1 A J2 ...) (J1 J2 A ... Jk) ... (J1 J2 ... Jk A))

and *REPT* takes its contents; as we saw before, =ITER= makes a list of elements of that form for each value of J.

--- see definition and examples in next page ---

```
  PERMUT   LISP      03/24  0651.3


DEFINE((
(PERMUTATIONS (LAMBDA (S) (CONVERT
()
(QUOTE(  X (XXX) (YYY)  ))
S
(QUOTE(  C1 (
( (X) ((X)) )
( (X XXX)  (=SKEL= A EXPR X (=ITER= J (=BEGN= (XXX)) (*REPT* (J ()) C2)
                                                              )))
) C2 (
( (() (YYY))  ((YYY A)) )
( ((X XXX) (YYY))  ((YYY A X XXX) (*REPT* ((XXX) (YYY X)) )) )
)))
))
```

```
r convrt
W 652.4
load ((permut))
 NIL

permutations ((1))
 ((1))

permutations ((1 2))
 ((1 2) (2 1))

permutations ((1 2 3))
 ((1 2 3) (2 1 3) (2 3 1) (1 3 2) (3 1 2) (3 2 1))

permutations ((a b c d))
 ((A B C D) (B A C D) (B C A D) (B C D A) (A C B D) (C A B D)
  (C B A D) (C B D A) (A C D B) (C A D B) (C D A B) (C D B A)
  (A B D C) (B A D C) (B D A C) (B D C A) (A D B C) (D A B C)
  (D B A C) (D B C A) (A D C B) (D A C B) (D C A B) (D C B A))

permutations (())
 NIL


stop
R 3.200+9.116
```

(*MARK X L) is intended to identify all the occurrences of the atom (or expression) X in the list L. This is done by leaving X as it is, but writing the atom * in place of any other subexpression. Thus, a sublist which contains no X's is replaced by a *, as are different atoms. However, the structure of a subexpression which is not X-free is detailled to the extent that all other subexpressions of the same level are indicated.

The decision as to whether an expression is X-free is made by the pattern XX, defined by (=OR= (=== X ===) (=== XX ===)) which matches any list which either directly contains an X, or contains a sublist with a similar property.

```
     DEFINE ((

(*MARK (LAMBDA (X L) (CONVERT
(CONS (QUOTE X) (CONS (QUOTE VAR) (CONS X
(QUOTE (
     XX      PAT      (=OR= (=== X ===) (=== XX ===))
     )))))
(QUOTE (
     ))
L
(QUOTE ((*0 (
     (X X)
     (XX (=ITER= J =SAME= (=REPT= J)))
     (== *)
     )))
     )))



     ))

r convrt
W 027.0
load ((mark))
 NIL
*mark (x (1 2 3 (1 2 3 (1 2 x (3 4) y x) x (((x) *) *) *) y y))
 (* * * (* * * (* * X * * X) X (((X) *) *) *) * *)

*mark (x ((((m (m (m x) m) m) (x x) m) m)))
 ((((* (* (* X) *) *) (X X) *) *))

*mark (0 (1 (2 3 4 x) 5 6 7 x) 8 9 0 0 0 0 0)
***  ERROR  A1LSA2
 ((X . 0) (L 1 (2 3 4 X) 5 6 7 X))
*mark (x (1 2 3 4 (1 2 3 (1 2 3 x 5 6 7) c c c)c x x))
 (* * * * (* * * (* * * X * * *) * * *) * X X)
stop
R 5.253+8.716
```

(FORMUL L) is a LISP function defined through CONVERT which will transform an algebraic formula written in infix notation to the binary prefix form. The argument L, which is the formula to be transformed, may use any variables, connected by the symbols PL (plus), MI (minus), TI (times), DI (divide), or PO (power), whose binding strength increases in the order mentioned. Thus the usual conventions for avoiding parentheses in associative or distributive configurations apply.

The CONVERT function implementing FORMUL is

```
(FORMUL (LAMBDA (L) (CONVERT
(QUOTE (
     LL       SKEL     (=REPT= (=WHEN= (LLL) (L) L))
     RR       SKEL     (=REPT= (=WHEN= (RRR) (R) R))
     ))
(QUOTE (
     L R (LLL) (RRR)
     ))
L
(QUOTE (*0 (
     ((LLL PL RRR) (PLU LL RR))
     ((LLL MI RRR) (MIN LL RR))
     ((LLL TI RRR) (TIM LL RR))
     ((LLL DI RRR) (DIV LL RR))
     ((LLL PO RRR) (POW LL RR))
     )))
     )))
```

The binding strength hierarchy is determined by the order in which the rules are written; thus no attempt will be made to locate a product will be made in any expression which contains a sum, and so on. Since CONVERT always makes leftmost matching fragments as small as possible, association is always made to the right; (A PL B PL C) would be transformed to (PLU A (PLU B C)). Any expression not connected by one of the five admissable algebraic connectors is left unchanged. The choice of the names of the connectors is occasioned by LISP 1.5's aversion for the pure algebraic signs.

The skeletons LL and RR serve to analyze the arguments of the algebraic connectives further as formulas, so that formulas may be formed recursively with formulas as subexpressions. Moreover, when the fragments LLL or RRR contain only one expression, it is necessary to avoid endowing them with a spurious pair of parentheses, wherefore the configuration (=WHEN= (LLL) (L) L).

```
r convrt
W 2337.0
load ((formul))
 NIL


formul ((x pl y pl z pl 7 ti x po 2))
 (PLU X (PLU Y (PLU Z (TIM 7 (POW X 2)))))


formul ((x ti (y pl z pl 28) po 2))
 (TIM X (POW (PLU Y (PLU Z 28)) 2))

formul (((x pl 2) ti (z pl y po 3) 1#pl 3 ti t))
 (PLU (TIM (PLU X 2) (PLU Z (POW Y 3))) (TIM 3 T))

formul (((x pl 2) ti (x pl z po 3) pl 3 ti t))
 (PLU (TIM (PLU X 2) (PLU X (POW Z 3))) (TIM 3 T))
```

CONVERT is well adapted to making transformations arising from the word problem. Given strings of characters from some alphabet, one may specify the equivalence of certain strings. Once done, it is reasonable to ask whether there is thereby implied the equivalence of a given pair of strings. In order to determine the existence of such an implied equivalence one might systematically seek out the various strings which are known to have equivalents, replacing them by their equivalent in order to transform one member of the pair into the other. In particular, one might attempt to reduce an arbitrarily given string to a canonical form. Whereas the determination of such equivalences is a generally insoluble problem, many classes of equivalences do in fact lead to determinate problems, and one may still be interested in seeing the effect of a given transformation strategy in the general case.

The two programs included here as examples will attempt the reduction of a word to canonical form; the strategy is to arrange the letters in lexicographic order by rules which describe an alternative form to two letters written in the wrong order; they also attempt to control the number of consecutive repetitions of the same letter, since one of the rules states that some such sequences are equivalent to the null string. Of course, these rules belong to an especially simple word problem.

(WORD W) makes such a reduction for the group defined by $A^3 = I$, $B^2 = I$, $BA = A^2B$, where powers are indicated by repeated atoms, the identity by the null string. ie, A A A is deleted from any string, as is B B, and B A is replaced by A A B.

(WORD* W) transforms strings according to the rules that A B A B = C, C C C, A A A A, and B B are deleted, C A = A B C C, C B = A C C, and B A A = A B.

In each case, the examples shown are derived by the inclusion of the rule (=PRI= =SAME=) as the first rule of the respective rule set. We thereby are able to see the step by step transformation of the input string to its final form.

```
(WORD (LAMBDA (W) (CONVERT
(QUOTE (
      ))
(QUOTE (
      (LLL) (RRR)
      ))
W
(QUOTE (*0 (
(=PRI= =SAME=)
      ((LLL A A A RRR) (=REPT= (LLL RRR)))
      ((LLL B B RRR) (=REPT= (LLL RRR)))
      ((LLL B A RRR) (=REPT= (LLL A A B RRR)))
      )))
      )))
```

```
(WORD* (LAMBDA (W) (CONVERT
(QUOTE (
      ))
(QUOTE (
      (LLL) (RRR)
      ))
W
(QUOTE ((*0 (
(=PRI= =SAME=)
      ((LLL A B A B RRR) (=REPT= (LLL C RRR)))
      ((LLL C C C RRR) (=KEPT= (LLL RRR)))
      ((LLL A A A A RRR) (=REPT= (LLL RRR)))
      ((LLL B B RRR) (=REPT= (LLL RRR)))
      ((LLL C A RRR) (=KEPT= (LLL A B C C RRR)))
      ((LLL C B RRR) (=REPT= (LLL A C C RRR)))
      ((LLL B A A RRR) (=REPT= (LLL A B RRR)))
      )))
```

```
clock(())  word* ((c c c c b b b b a a a a))  clock (t)
 0
 (C C C C B B B B A A A A)
 (C B B B B A A A A)
 (C B B B B)
 (C B B)
 (C)
 (C)
 3
```

```
clock (())   word* ((c c b a a a c c b a a a c c b a a a )) clock (t)
 0
 (C C B A A A C C B A A A C C B A A A)
 (C A C C A A A C C B A A A C C B A A A)
 (A B C C C C A A A C C B A A A C C B A A A)
 (A B C A A A C C B A A A C C B A A A)
 (A B A B C C A A C C B A A A C C B A A A)
 (C C C A A C C B A A A C C B A A A)
 (A A C C B A A A C C B A A A)
 (A A C A C C A A A C C B A A A)
 (A A A B C C C C A A A C C B A A A)
 (A A A B C A A A C C B A A A)
 (A A A B A B C C A A C C B A A A)
 (A A C C C A A C C B A A A)
 (A A A A C C B A A A)
 (C C B A A A)
 (C A C C A A A)
 (A B C C C C A A A)
 (A B C A A A)
 (A B A B C C A A)
 (C C C A A)
 (A A)
 (A A)
 39
```

```
clock (()) word* ((b c a b b c c a a b c c a b a b c a b c c a a a))
                                                        clock (t)
  0
  (B C A B B C C A A B C C A B A B C A B C C A A A)
  (B C A B B C C A A B C C C C A B C C A A A)
  (B C A B B C C A A B C A B C C A A A)
  (B C A C C A A B C A B C C A A A)
  (B A B C C C C A A B C A B C C A A A)
  (B A B C A A B C A B C C A A A)
  (B A B A B C C A B C A B C C A A A)
  (B C C C A B C A B C C A A A)
  (B A B C A B C C A A A)
  (B A B A B C C B C C A A A)
  (B C C C B C C A A A)
  (B B C C A A A)
  (C C A A A)
  (C A B C C A A)
  (A B C C B C C A A)
  (A B C C B C A B C C A)
  (A B C C B A B C C B C C A)
  (A B C C B A B C C B C A B C C)
  (A B C C B A B C C B A B C C B C C)
  (A B C A C C A B C C B A B C C B C C)
  (A B A B C C C C A B C C B A B C C B C C)
  (C C C C C A B C C B A B C C B C C)
  (C C A B C C B A B C C B C C)
  (C A B C C B C C B A B C C B C C)
  (A B C C B C C B C C B A B C C B C C)
  (A B C A C C C C B C C B A B C C B C C)
  (A B C A C B C C B A B C C B C C)
  (A B A B C C C C B C C B A B C C B C C)
  (C C C C B C C B A B C C B C C)
  (C B C C B A B C C B C C)
  (A C C C C B A B C C B C C)
  (A C B A B C C B C C)
  (A A C C A B C C B C C)
  (A A C A B C C B C C B C C)
  (A A A B C C B C C B C C B C C)
  (A A A B C A C C C B C C B C C)
  (A A A B C A C B C C B C C)
  (A A A B A B C C C B C C B C C)
  (A A C C C C B C C B C C)
  (A A C B C C B C C)
  (A A A C C C C B C C)
  (A A A C B C C)
  (A A A A C C C C)
  (A A A A C)

  (C)
  (C)
  74
```

```
r convrt
W 1652.5
load ((word))
 NIL

word ((a b a b a a b b b a b a a b b))
 (A B A B A A B B B A B A A B B)
 (A B A B A A B A B A A B B)
 (A B A B A A B A B A A)
 (A A A B B A A B A B A A)
 (B B A A B A B A A)
 (A A B A B A A)
 (A A A A B B A A)
 (A B B A A)
 (A A A)
 NIL
 NIL


word ((b b b b a a a a a a a))
 (B B B B A A A A A A A)
 (B B B B A A A A)
 (B B B B A)
 (B B A)
 (A)
 (A)


word ((b b b b b a a a a a a a ))
 (B B B B B A A A A A A A)
 (B B B B B A A A A)
 (B B B B B A)
 (B B B A)
 (B A)
 (A A B)
 (A A B)

clock (())   word ((b a a b a a b a a b a a b a a b a a))  clock (t)
 0
 (B A A B A A B A A B A A B A A B A A)
 (A A B A B A A B A A B A A B A A B A A)
 (A A A A B B A A B A A B A A B A A B A A)
 (A B B A A B A A B A A B A A B A A)
 (A A A B A A B A A B A A B A A)
 (B A A B A A B A A B A A)
 (A A B A B A A B A A B A A)
 (A A A A B B A A B A A B A A)
 (A B B A A B A A B A A)
 (A A A B A A B A A)
 (B A A B A A)
 (A A B A B A A)
 (A A A A B B A A)
 (A B B A A)
 (A A A)
 NIL
 NIL
 18
```

Sometimes it is more convenient to use CONVERT to construct a special purpose program to handle a given problem than to try to program the problem itself in its full generality. An example of such a situation is the program (PERGEN S) designed to list all the permutations of the set S. The underlying idea is quite simple --- to permute the elements of S we map its first element into any other, including itself. We then map the second element into one of those remaining, then the third element into some unused image, and so on. There are continually fewer alternatives available until finally the image of the last element is determined fully.

Assuming that the elements of S are atomic, we may use them as skeletons, to be replaced by their images under the permutation. In order to assign their images, as well as to vary them in a systematic manner, we use the control skeleton =ITER=; but constructed in advance so as to use the elements of S as its indices, and to cause them to vary through suitably restricted subsets, as the choice of the value of one index restricts the range available to its successors.

In the examples shown, the generated skeleton is printed before it is executed, which allows one to see how the procedure described above has been implemented. Since a generative stage is involved, various skeleton names such as *ITER* must be quoted, so that they will not be replaced at the time the skeleton containing them is constructed; this is done by introducing synonymns for them in the EXPR mode, :IT:, for example.

If we examine the example PERGEN ((A B C)), we see that the intermediate skeleton which is generated is

```
((*ITER* A (=COMP= (=QUOT= (A B C)) NIL) (*ITER* B (=COMP=
(=QUOT= (A B C)) (A)) (*ITER* C (=COMP= (=QUOT= (A B C)) (A
B)) (A B C))))
```

The outer iteration allows the variable A to range through the values A, B, or C. The second iteration allows the variable B to range through the set (=COMP= (=QUOT= (A B C)) (A)). We must recall that A is now a variable, so that on the three different occasions that we will execute the second *ITER*, A will have three different values, and hence the complement, calculated during the execution time, will be successively (B C), (A C), and (A B). In examining the result,

```
((A B C) (A C B) (B A C) (B C A) (C A B) (C B A))
```

we see that A varies most slowly, taking values A, B, C; then B varies through B, C; then A, C; finally A, B, while C varies most rapidly, which is unnoticed since its value is forced.

```
r convrt
W 746.8
load ((pergen))
 NIL

pergen ((a b c))
 ((*ITER* A (=COMP= (=QUOT= (A B C)) NIL) (*ITER* B (=COMP=
 (=QUOT= (A B C)) (A)) (*ITER* C (=COMP= (=QUOT= (A B C)) (A
 B)) (A B C)))))
 ((A B C) (A C B) (B A C) (B C A) (C A B) (C B A))


pergen ((a b c d))
 ((*ITER* A (=COMP= (=QUOT= (A B C D)) NIL) (*ITER* B (=COMP=
 (=QUOT= (A B C D)) (A)) (*ITER* C (=COMP= (=QUOT= (A B C D))
 (A B)) (*ITER* D (=COMP= (=QUOT= (A B C D)) (A B C)) (A B C
 D))))))
 ((A B C D) (A B D C) (A C B D) (A C D B) (A D B C) (A D C B)
 (B A C D) (B A D C) (B C A D) (B C D A) (B D A C) (B D C A)
 (C A B D) (C A D B) (C B A D) (C B D A) (C D A B) (C D B A)
 (D A B C) (D A C B) (D B A C) (D B C A) (D C A B) (D C B A))


stop
R 4.966+5.733


print pergen lisp
W 748.6

PERGEN   LISP       03/24  0748.7

       DEFINE((
(PERGEN (LAMBDA (S) (CONVERT
(QUOTE (
     TT        PAV      (T TTT)
    :IT:       EXPR      *ITER*
    :CO:       EXPR      =COMP=
    :QU:       EXPR      *QUOT=
    ))
    (QUOTE (
         T (TTT)
         ))
    S
    (QUOTE (*1 (
    (== (=SKEL= S EXPR *SAME= (*SKEL= Q SKEL (=PRNT= ((=REPT= S *2 (
         (TT (:IT: T (:CO: (:QU: S) (=COMP= S TT)) (*REPT= (TTT))))
         (TT (:IT: T (:CO: (:QU: S) (:QU: (=COMP= S TT))) (=REPT= (TTT))))
         (() S)
         )))) Q)))
         )))
)))

    ))
    R .816+.333
```

The Poisson Bracket, [f, g] of two functions, $f(p_i, q_i)$ and $g(p_i, q_i)$ is defined by the formula

$$[f, g] = \sum_{i=1}^{n} \frac{\partial f}{\partial q_i}\frac{\partial g}{\partial p_i} - \frac{\partial f}{\partial p_i}\frac{\partial g}{\partial q_i}$$

However, it is possible to give an axiomatic treatment to the Poisson Bracket by noticing that it is an alternating bilinear functional which is also a derivative. That is to say, it obeys the rules

1. $[f, g] = -[g, f]$
2. $[af + bg, h] = a[f, h] + b[g, h]$
3. $[fg, h] = f[g, h] + [f, h]g$

as well as the normalization conditions

$[p_i, p_j] = 0$
$[q_i, p_j] = 1$ if $i = j$; 0 otherwise
$[q_i, q_j] = 0$.

It is therefore possible to program the calculation of the Poisson Bracket of polynomials in the coordinates q_i and momenta p_i as a symbol manipulation program effecting the transformations implied by the axioms. Unfortunately the axioms in their simplest form introduce a great number of spurious factors and summands which are either 0 or 1, or equivalent to them after some adjustment. It is therefore desirable to incorporate an algebraic simplification program with the Poisson Bracket calculation at least powerful enough to counteract the trivial factors and summands introduced by the differentiation process.

This simplification occupupies the greater part of the volume of the program. For example, sums are simplified by removing zero summands, summing numerical summands, bringing numerical summands to the leftmost position, bringing negative signs to the outermost level, applying the associative law to write multiple sums without parentheses, and trying to improve numerical factors whereever possible. differences and negatives are principally simplified by eliminating zero arguments and computing numerical negatives whereever possible. Product simplification is very similar to sum simplification; a product containing a numerical zero factor is at once set equal to zero, factors of one are excluded, numerical factors are multiplied, preference is given to the power representation, and so on. The trivial cases of exponents are reduced.

Input to the program (PB X Y) is in the form of polynomials written in infix notation with the algebraic connectors PL (plus), MI (minus), TI (times), and PO (power). Conjugate variable-momenta pairs consist of an atom, as X to represent the coordinate, and the same atom inclosed in parentheses, as (X), to represent the conjugate momentum. Intermediate calculations are done in prefix functional form, but the output is again presented in infix notation. Little simplification not occesioned by the derivation process is made, thus a negative pair of terms which are separated, or which differ in the internal arrangement of their subexpressions will not be replaced by zero, for example.

```
print *pb* lisp
W 0626.8

  *PB*   LISP      03/25  0626.8


      DEFINE ((
(PB (LAMBDA (X Y) (CONVERT
(QUOTE (
      =X=       PAV       =ATO=
      =Y=       PAV       =ATO=
      =M=       UNO       ((M1 X) Y)
      =N=       UNO       (=K= X)
      =K=       PAV       =NUM=
      =L=       PAV       =NUM=
      XX        SKEL      (=WHEN= (XXX) (X) X)
      YY        SKEL      (=WHEN= (YYY) (Y) Y)
      ZZ        SKEL      (=WHEN= (ZZZ) (Z) Z)
      PLU       REPT      ( ((== (=REPT= (=WHEN= =SAME= (=N= =N=) (=K= X)) *P (
                           ((0 X) X)
                           ((=K= =L=) (=PLUS= =K= =L=))
                           ((=UNO= (=K= (=L= PL ZZZ))) (=REPT= ((=PLUS= =K= =L=) ZZ)))
                           ((X (=K= PL Y)) (=K= PL (=REPT= (X Y))))
                           ((=M= =M=) (MIN Y X))
                           ((X (Y PL ZZZ)) (X PL Y PL ZZZ))
                           (((=K= TI X) (=L= TI X)) ((=PLUS= =K= =L=) TI X))
                           ((X X) (2 TI X))
                           ((X Y) (X PL Y))
                          ))))
      MIN       REPT      (
                           ((0) 0)
                           ((=K= =L=) (=MINS= =K= =L=))
                           (((M1 X)) X)
                           (((=K= TI ZZZ)) (TIN (=MINS= 0 =K=) ZZ))
                           ((X 0) X)
                           ((0 X) (M1 X))
                           ((X X) 0)
                           ((X =K=) (M1 (=K= M1 X)))
                           ((X Y) (X M1 Y))
                           ((=K=) (=MINS= 0 =K=))
                           ((X) (M1 X))
                          )

                                                                                25
```

```
TIM     REPT    ( ((== (=REPT= (=WHEN= =SAME= (=N= =N=) (=K= X)) *T (
                  ((=== 0 ===) 0)
                  ((1 X) X)
                  ((=K= =L=) (=TIMS= =K= =L=))
                  ((=UNO= (=K= (=L= TI ZZZ))) (=REPT= ((=TIMS= =K= =L=) ZZ)))
                  ((X (=K= TI ZZZ)) (=K= TI (=REPT= (X ZZ))))
                  ((=M= =N=) (MIN (=REPT= (X Y))))
                  (((X PO =K=) (X PO =L=)) (X PO (=PLUS= =K= =L=)))
                  ((X (X PO =K=)) (X PO (=INCR= =K=)))
                  ((X X) (X PO 2))
                  ((=UNO= (X (Y TI ZZZ))) (X TI Y TI ZZZ))
                  ((X Y) (X TI Y))
                 ))))
POW     REPT    (
                  ((== 0) 1)
                  ((1 ==) 1)
                  ((X 1) X)
                  ((X Y) (X PO Y))
                 )
     ))
(QUOTE (
     X Y Z (XXX) (YYY) (ZZZ)
     ))
(LIST X Y)
(QUOTE (*9 (
     ((=X= (=X=))   1)
     ((=X= (=Y=))   0)
     ((=X= =Y=)     0)
     (((=X=) (=Y=)) 0)
     (((XXX PL YYY) Z) (PLU (=REPT= (XX Z)) (=REPT= (YY Z))))
     (((XXX MI YYY) Z) (MIN (=REPT= (XX Z)) (=REPT= (YY Z))))
     (((XXX TI YYY) Z) (PLU (TIM XX (=REPT= (YY Z))) (TIM (=REPT= (XX Z)) YY)))
     (((X PO Y) Z)     (TIM Y (TIM (POW X (MIN Y 1)) (=REPT= (X Z)))))
     ((X Y)            (MIN (=REPT= (Y X))))
     )))
     )))
```

```
r convrt
W 631.1
load ((*pb*))
 NIL
excise (t)
 NIL
clock (()) pb (((x) po 2 pl (y) po 2 pl x po 2 pl y po 2)  ((x) ti y pl x ti (y))) clock (t)
 0
 ((-2 TI (Y) TI (X)) PL (-2 TI (X) TI (Y)) PL (2 TI X TI Y)
 PL (2 TI Y TI X))
 82


clock (()) pb (((x) ti y pl x ti (y)) ((x) ti (y) pl x ti y)) clock (t)
 0
 ((((X) PO 2) MI (Y PO 2)) PL (((Y) PO 2) MI (X PO 2)))
 65

clock (()) pb (((x) po 2 pl (y) po 2 pi x po 2 pl y po 2) (x pl y)) clock (Y#T)
 0
 ((-2 TI (X)) PL (-2 TI (Y)))
 36


clock (()) pb (((x) po 17 pl y) (x ti (y))) clock (T)
 0
 ((-17 TI (Y) TI ((X) PO 16)) PL X)
 28




stop
R 25.516+9.016
```

A rat, upon being introduced to a new maze, is immediately interested in learning all the paths leading between various pairs of points. The interest of a mathematician is more modest and yet more systematic, since once a program is given for finding all the paths between one pair of points, that pair being arbitrary, there exist known methods for solving the general case. To find all paths between a pair of points, we may suppose that the points are in fact identical, and that moreover, we are not interested in loops which lead from a point to itself. Such not being the case, it would be possible that we have a pair of points which are directly linked, through a primitive path. Neither being the case, we might consider the image set of our initial point, by which we mean the set of all points linked to the initial point by a non-null, primitive path, and the counterimage set of the final point, meaning the set of points linked to the final point by a single primitive path. If we form the cartesian product of the image set and the counterimage set, and suppose known all paths linking those pairs of points --- a point in the image set one step away from the initial point to a point in the counterimage set, likewise one step away from the final point --- we then have a recursive solution to the original problem.

For mathematical --- or computer --- consideration we must have a representation of the maze which is to be studied, and this is conveniently given by a list of the primitive links, the pair of points (X Y) belonging to the link list L if a path joins X to Y. We may suppose this is a directed path, and insist that (Y X) also appear in the list if the path is bidirectional.

A pair of points is then connected if there exist links (A X1) (X1 X2) ... (Xi Xi+1) ... (Xn B) all belonging to the list L, A and B being the initial and final point respectively. Such a link may exist in one direction but not the other.

An effective way to avoid loops in enumerating the paths is to remove the initial and final points from consideration in the inductive step, since any path arriving eventually at the initial point must form a loop, as well as a path to the final point initiating from the final point.

In the program (PATHS A B M), A is the initial point, B is the final point, and M is the link list. A and B are variables within the program enjoying a similar significance. (LLL) and (RRR) are fragments retaining the left and right halves of a list which we analyze. The bucket variables are respectively A* which collects the elements of the image set, B* which collects the elements of the counterimage set, and X which collects links not originating or terminating from A or B. The fragment pattern (UU)

((*OR* ((A A*) UU) ((= A) UU) ((B* B) UU) ((B =*) UU) (X UU) ()))

is used to see the link list decomposed by the bucket variables; X will become the new link list in the recursive subproblem.

The key rule is

((A B (UU)) (=ITER= I A* J B* (K) (=REPT= (I J X)) (A K B)))

which sees the initial and final point together with the link list decomposed into the image set, counterimage set, and link list from which the initial and final points have been eliminated. For each pair of elements from A* and B*, the process is repeated, and to the path list --- a list of points forming the joining path --- is appended the outer points A and B.

This rule defines the the repetitive condition of the CONVERT program; the terminal conditions arise when we encounter a pair of points which turn out to be in fact the same, or a pair which are directly linked. In the latter case we search for any additional indirect links.

Should we eventually exhaust the link list without the ends joining, we produce a null fragment which, by producing a vacuous index set for =ITER=, causes the tentatively formed chain to be discarded.

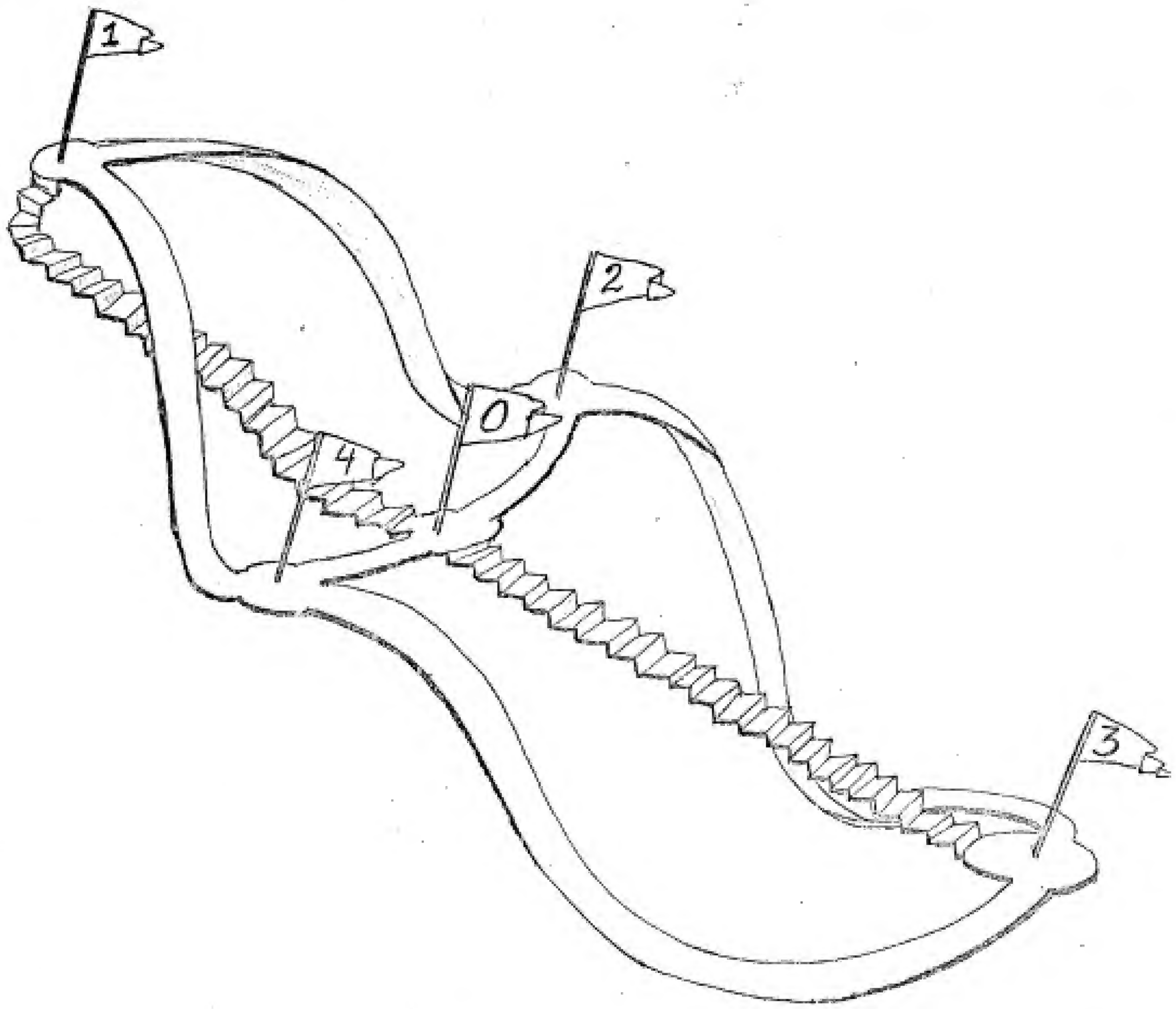


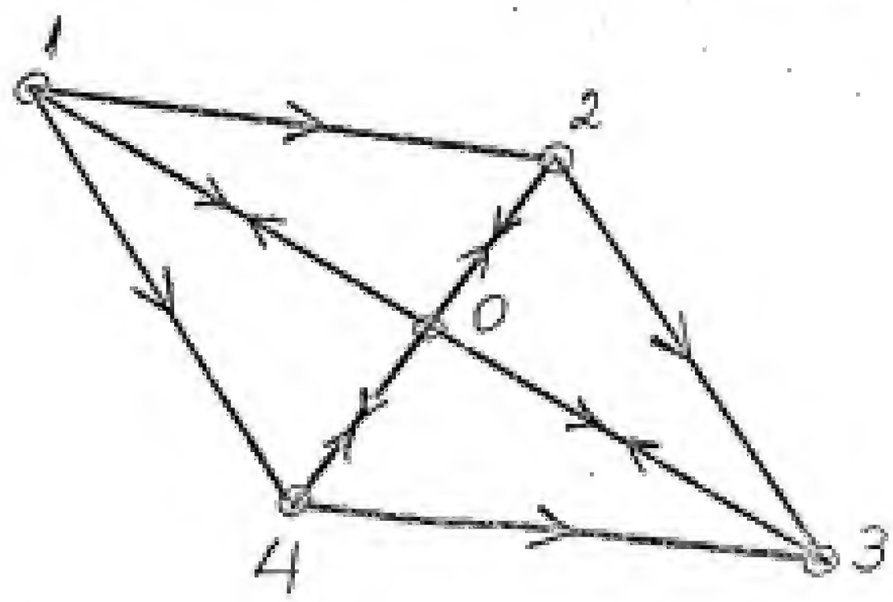
Fig. 'T R I A L'.
One of the problems solved by the function (PATHS A B M).

```
(PATHS (LAMBDA (A B M) (CONVERT
(QUOTE (
(UU) PAT ((*OR* ((A A*)UU) ((== A)UU) ((B* B)UU) ((B ==)UU) (X UU) ()))
 A*   BUV ==
 B*   BUV ==
 X    BUV ==
))
(QUOTE ( A B (LLL) (RRR) ))
(LIST A B M)
(QUOTE ( *0 (
( (A A ==) ((A)) )
( (A B (LLL (A B) RRR)) ((A B) (*REPT* (A B (LLL RRR)))) )
( (A B (UU)) (=ITER= I A* J B* (K) (=REPT= (I J X)) (A K B)) )
( == () )
)))
)))
```

```
CSET (TRIAL ((0 1) (0 2) (0 3) (0 4) (1 0) (2 0)
            (3 0) (4 0) (1 2) (1 4) (2 3) (4 3)))
```

Fig. 'T R I A L'.

Not all the links are bidirectional.




```
r convrt
W 753.3
load ((paths))
 NIL

clock (())   e (paths 1 3 trial)    clock (t)
 0
 ((1 0 3) (1 0 2 3) (1 0 4 3) (1 2 0 3) (1 2 3) (1 2 0 4 3)
 (1 4 0 3) (1 4 0 2 3) (1 4 3))
 33


clock (())   e (paths 3 1 trial)    clock (t)
 0
 ((3 0 1))
 12

clock (())   e (paths 2 4 trial)    clock (t)
 0
 ((2 0 4) (2 0 1 4) (2 3 0 4) (2 3 0 1 4))
 18
```

```
CSET (MESH  ((A C) (C B) (B C) (D E) (D H) (D F)
            (E F) (E G) (E H) (F H) (F G) (F D)
      (G D) (G H) (G E) (H D) (H E) (H G) (H F)
                                          I () ))
```

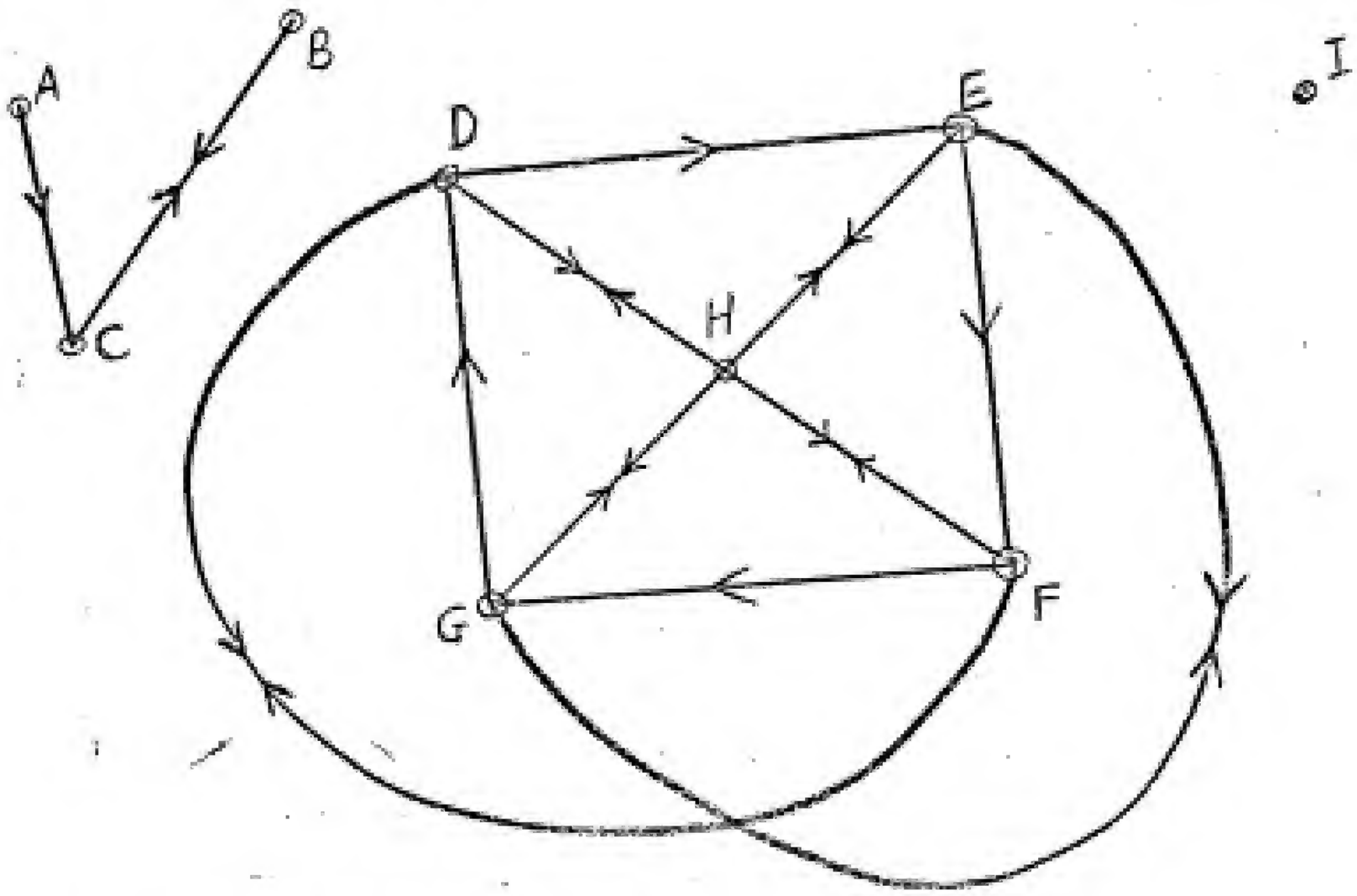


Fig. 'M E S H'.

```
clock (())  e (paths (quote a) (quote b) mesh)  clock (t)
 0
 ((A C B))
 18

clock (())  e (paths (quote b) (quote a) mesh)  clock (t)
 0
 NIL
 17

clock (())  e (paths (quote c) (quote h) mesh)  clock (t)
 0
 NIL
 38

clock (())  e (paths (quote i) (quote g) mesh)  clock (t)
 0
 NIL
 16

clock (())  e (paths (quote d) (quote f) mesh)  clock (t)
 0
 ((D F) (D E F) (D E H F) (D E G H F) (D H E F) (D H G E F)
 (D H F))
 33

clock (())  e (paths (quote f) (quote d) mesh)  clock (t)
 0
 ((F D) (F H G D) (F H E G D) (F H D) (F G D) (F G H D) (F G
 E H D))
 33
```

```
clock (())  e (paths (quote h) (quote f) mesh)
 0
 ((H F) (H D F) (H D E F) (H E G D F) (H E F) (H G D F) (H G
                                                  E F) (H G D E F))
  clock (())  e (paths (quote h) (quote g)  mesh)  clock (t)
    0
   ((H G) (H D E G) (H D F G) (H D E F G) (H E G) (H E F G) (H
 F D E G) (H F G))
                39

                CSET (NET ((A E) (A F) (E A) (E K) (E C) (E F) (K E)
                           (K P) (K F) (C E) (C M) (C D) (D C) (D P)
              (D F) (P K) (P D) (P L) (L P) (L B) (L M) (M L) (M Y)
              (M C) (Y M) (Y B) (F A) (F E) (F K) (F D) (B L) (B Y)))
```

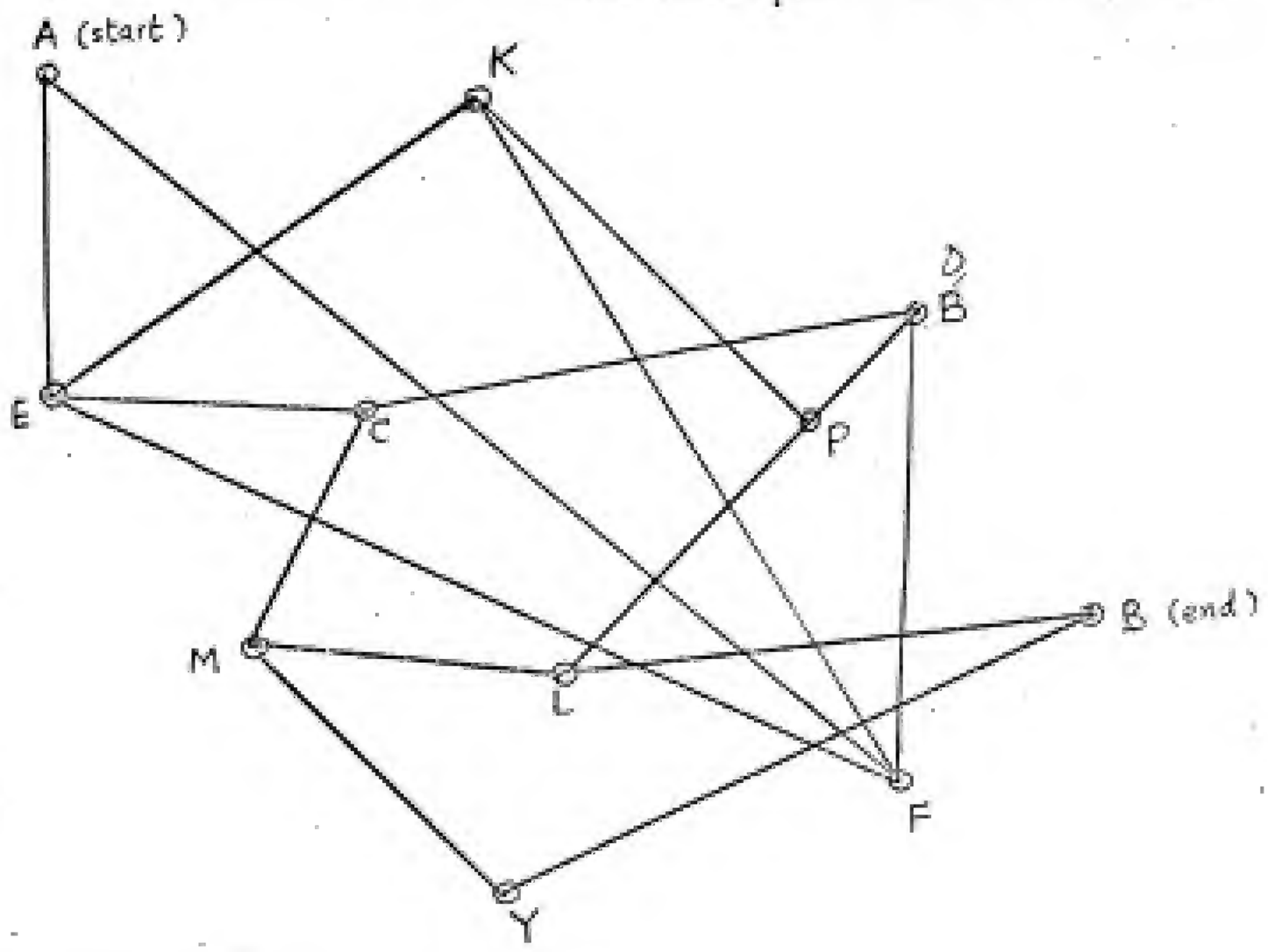


Fig. 'N E T'.

```
excise (t)
clock (())  e (paths (quote a) (quote b) net)  clock (t)
 NIL
 0
 ((A E K P L B) (A E K F D P L B) (A E K P D C M L B) (A E K
 F D C M L B) (A E C D F K P L B) (A E C D P L B) (A E C M L
 B) (A E F K P L B) (A E F D P L B) (A E F K P D C M L B) (A
 E F D C M L B) (A E K P D C M Y B) (A E K P L M Y B) (A E K
 F D C M Y B) (A E K F D P L M Y B) (A E C M Y B) (A E C D P
 L M Y B) (A E C D F K P L M Y B) (A E F K P D C M Y B) (A E
 F K P L M Y B) (A E F D C M Y B) (A E F D P L M Y B) (A F E
 K P L B) (A F E C D P L B) (A F E K P D C M L B) (A F E C M
 L D) (A F K P L B) (A F K E C D P L B) (A F K E C M L B) (A
 F K P D C M L B) (A F D P L B) (A F D C E K P L B) (A F D C
 M L B) (A F D P K E C M L B) (A F E K P D C M Y B) (A F E K
 P L M Y B) (A F E C M Y B) (A F E C D P L M Y B) (A F K E C
 M Y B) (A F K E C D P L M Y B) (A F K P D C M Y B) (A F K P
 L M Y B) (A F D C M Y B) (A F D C E K P L M Y B) (A F D P K
 E C M Y B) (A F D P L M Y B))
 501
```

```
print paths lisp
W 216.8

 PATHS   LISP       04/02  0216.8



      DEFINE ((
(PATHS (LAMBDA (A B M) (CONVERT
(QUOTE (
(UU) PAT ((*OR* ((A A*)UU) ((== A)UU) ((B* B)UU) ((B ==)UU) (X UU) ()))
 A*   BUV ==
 B*   BUV ==
 X    BUV ==
))
(QUOTE ( A B (LLL) (RRR) ))
(LIST A B M)
(QUOTE ( *0 (
( (A A ==) ((A)) )
( (A B (LLL (A B) RRR)) ((A B) (*REPT* (A B (LLL RRR)))) )
( (A B (UU)) (=ITER= I A* J B* (K) (=REPT= (I J X)) (A K B)) )
( == () )
)))
)))
 ))
```

```
CSET (TRIAL ((0 1) (0 2) (0 3) (0 4) (1 0) (2 0)
             (3 0) (4 0) (3 2) (1 4) (2 3) (4 3)))

CSET ('NET ((A E) (A F) (E A) (E K) (E C) (E F) (K E)
            (K P) (K F) (C E) (C M) (C D) (D C) (D P)
(D F) (P K) (P D) (P L) (L P) (L B) (L M) (M L) (M Y)
(M C) (Y M) (Y B) (F A) (F E) (F K) (F D) (B L) (B Y)))

CSET ('MESH  ((A C) (C B) (B C) (D E) (D M) (D F)
              (E F) (E G) (E M) (F M) (F G) (F D)
       (G D) (G M) (G E) (M D) (M E) (M G) (M F)
        I () ))
R 1.150+.556
```

Polynomial manipulation is complicated by the fact that a given polynomial can be written in a wide variety of equivalent forms; and it is therefore convenient to reduce a polynomial to some convenient standard form before using it in further calculations. (POLY) is a function, defined in CONVERT, for making such a reduction, and some further simple manipulations. It is a function of no variables because it reads in instructions, and data, from the teletype console.

We first define a polynomial; this is done in terms of =P= which recognizes polynomials, and *P*, which recognizes polynomial fragments. A polynomial is defined recursively as one of the following

a number
the variable, x
two fragments connected by a plus sign
two fragments connected by a minus sign, such that
the right fragment itself contains no minus signs
minus a polynomial
two fragments connected by a product sign
a polynomial to a positive power

In its turn a polynomial fragment is either

a polynomial
a sum of two fragments
a difference of two fragments, the second containing no minus
a product of two fragments

Allowing polynomial fragments permits one to make use of the associative properties of addition and multiplication by writing multiple sums or products without explicit parentheses to specify the association. The rules above cause a right association in testing a polynomial, except for minus. Although not ordinarily an associative operation, convention holds that (a - b - c - d) should be (((a - b) - c) - d), or a - (b + c + d). In splitting a polynomial into two fragments separated by a minus sign, it is thus necessary to see that the right fragment does not contain a minus. In such a context a monary minus will always cause confusion if it is not parenthesized when it conflicts with a product; ie we do not allow x * - y.

Although the =OR= defining a polynomial resembles the so-called "Backus Normal Form" it must be remembered that the alternatives of the =OR= are sought in order, so that a polynomial is decomposed as a product only if it cannot be decomposed as a sum or as a difference. The order in which the algebraic connectors are written therefore determines their binding strength, and on this account parentheses may be omitted when they would enclose terms joined by the stronger connective.

Similar in operation is the function NOR which reduces a polynomial to its internal representation, which is a list of its coefficients when written in the form

$$p(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots + a_nx^n.$$

Its representation would then be

$$(a_0\ a_1\ a_2\ a_3 \ldots\ a_n).$$

The operations which one makes for polynomials may be made for the internal representation. we have

PLU sums two polynomials by summing corresponding coefficients; if they are of different degrees the high order coefficients are appended to the sum of the common coefficients.

MIN as a monary operation, multiplies each coefficient by -1, and as a binary operation, works analogously to PLU.

TIM multiplies two polynomials by the rule $(a_0 + a_1 x\ p'(x))q(x) = a_0 q(x) + a_1 x\ p'(x)q(x)$, where $p'(x)q(x)$ is computed recursively. Multiplication by a constant is effected by multiplying each coefficient of q by that constant, and multiplication by x is effected by shifting --- adjoining a zero to the front of the coefficient list.

POW is realized by repeated multiplication, and no effort at optimization is made by making a binary decomposition of the power.

The terminal cases are the variable, which is replaced by (0 1), and a constant, which is replaced by (a). Reduction to the form of the internal representation is then effected by recognizing the possible combinations which fulfil the definition of a polynomial, and performing the indicated arithmetic with respect to the representation-list.

The functions PRD and PWR serve to convert the internal representation into a regular polynomial form. PWR writes the standard sum-of-powers according to which polynomials are generally written, and contains provision to write a in place of $a*x^0$, and b*x in place of $b*x^1$, as well as writing x^k in place of $1*x^k$, which is standard practice. The function PRD transforms the list $(a_0\ a_1\ \ldots\ a_n)$ to the product form

$$p(x) = (a_0 + x*(a_1 + x*(a_2 + \ldots\ x*(a_{n-1} + x*a_n)\ \ldots)))$$

which is preferable for evaluation of a polynomial because it minimizes the amount of multiplication. A similar decomposition is used by the function VAL, which evaluates the polynomial at a designated point.

Finally, the execution of the function (POLY) is controlled by a program containing a read loop. The admissable instructions are very simple, and could be considerably extended to perform a variety of polynomial operations.

E a evaluates the polynomial in the workspace at the point a

I p places the polyngmial p, input from the console, in the workspace. p may actually be any quantity.

N form the internal representatiou of the polynomial in the workspace, which replaces the workspace.

O print the workspace.

S terminate the program.

X e execute the expression e, input from the console, which may be any convert skeleton admissable to (POLY).

Y output the workspace in the polynomial sum-of-powers form; actually, the workspace assumed to be in internal representation, is transformed to this form and placed in the workspace, and must b output with O.

Z transform the workspace to the economized form, assuming that the workspace contains the internal list form.

TS place the workspace in temporary storage.

(TS) place temporary storage in the workspace.

Any command given, if it is not atomic and in the form of a list will be treated as a series of commands to be executed, and the same interpretation will be applied to the list elements in turn. Thus, (i n o) p will read p, change it to internal form, and write this form.

This program performs only the rudiments of polynomial manipulation; one could think of extending it to the generation of interpolation polynomials, numerical integration and differentiation formulas, and diverse other applications.

```
r convrt
W 135.0
load ((poly))        load the file poly
 NIL
poly ()              start poly
 ==
i ((x mi 1) po 7)    input (x-1)^7
 ==
o                    output
 ((X MI 1) PO 7)
 ==
n                    convert it to normal form
 ==
o                    write it out
 (-1 7 -21 35 -35 21 -7 1)   <-- normal form of (x-1)^7
 ==                          is -1+7x-21x^2+35x^3-....
(ts)                 store this result in Temporary Storage.
 ==
y                    convert it to sum of powers form
 ==
o                    print it
 (-1 PL 7 TI X PL -21 TI X PO 2 PL 35 TI X PO 3 PL -35 TI X
 PO 4 PL 21 TI X PO 5 PL -7 TI X PO 6 PL X PO 7)
 ==
ts                   bring the quantity from temporary storage to the workspace
 ==
z                    convert to economized form
 ==
o                    print the workspace
 (-1 PL X TI (7 PL X TI (-21 PL X TI (35 PL X TI (-35 PL X TI
 (21 PL X TI (-7 PL X TI 1))))))
 ==
```

```
i ((x pl 1) po 5 pl (x mi 1) po 5)
 ==
o
 ((X PL 1) PO 5 PL (X MI 1) PO 5)
 ==
n

 ==
o
 (-0 10 -0 20 -0 2)
 ==
((ts) y o ts z o)
 (10 TI X PL 20 TI X PO 3 PL 2 TI X PO 5)
 (-0 PL X TI (10 PL X TI (-0 PL X TI (20 PL X TI (-0 PL X TI
 2)))))
 ==
```

$(x+1)^5 + (x-1)^5$

$-0 + 10x - 0x^2 + 20x^3 + 2x^5$

this is a chain of commands

A new example:

```
r convrt
W 143.2
load ((poly))
 NIL
poly ()
 ==
i ((1 pl x pl x po 2) po 3)
 ==
n
 ==
o
 (1 3 6 7 6 3 1)
 ==
(ts)
 ==
(y o ts z o ts e o)
 (1 PL 3 TI X PL 6 TI X PO 2 PL 7 TI X PO 3 PL 6 TI X PO 4 PL
 3 TI X PO 5 PL X PO 6)
 (1 PL X TI (3 PL X TI (6 PL X TI (7 PL X TI (6 PL X TI (3 PL
 X TI 1)))))))
7
 185193          ← value at x=7
 ==
i (mi (x pl 5 ti x ti 8 ti x po 2 mi 9))
 ==
(o n o (ts) y o ts z o)
 (MI (X PL 5 TI X TI 8 TI X PO 2 MI 9))
 (9 -1 0 -40)
 (9 PL -1 TI X PL -40 TI X PO 3)
 (9 PL X TI (-1 PL X TI (0 PL X TI -40)))
 ==
(ts e o) 99
 -38812050
 ==
(ts e o) 2.607912
 -0.75324958E3
 ==
s
 (THE GRASS IS GREENER)
```

store in temp storage, evaluate it (at x = 99) and output it

```
      DEFINE ((
(POLY (LAMBDA () (CONVERT
(QUOTE (
      =N=       PAV       =NUM=
      =V=       VAR       X
     =POS=      PAT       =NUM=
     (WWW)      PAV       (($AND$ (*W*) (VVV)))
     (*W*)      PAT       ((*NOT* (=== MI ===)))
      =P=       PAT       (=OR= =NUM= =V= (*P* PL *P*) (*P* MI *W*) (MI =P=) (*P* TI *P*)
                                                                    (=P= PO =POS=))
     (*P*)      PAT       ((*OR* (=P=) (*P* PL *P*) (*P* MI *W*) (*P* TI *P*)))
      UU        SKEL      (=COND= (UUU) (U) U)
      VV        SKEL      (=COND= (VVV) (V) V)
      NOR       REPT      (((U) (=REPT= U *2 (
                           (=N= (=SAME=))
                           (=V= (0 1))
                           ((UUU PL VVV) (PLU (=REPT= UU) (=REPT= VV)))
                           ((MI U) (MIN (=REPT= U)))
                           ((UUU MI WWW) (MIN (=REPT= UU) (=REPT= VV)))
                           ((UUU TI VVV) (TIM (=REPT= UU) (=REPT= VV)))
                           ((UUU DI VVV) (DIV (=REPT= UU) (=REPT= VV)))
                           ((U PO =N=) (POW (=REPT= U) =N=))
                          ))))
PLU       REPT      (
                           (((U UUU) (V VVV)) ((=PLUS= U V) (*REPT* ((UUU) (VVV)))))
                           ((() ()) ())
                           ((=OR= (U ()) (() U)) U)
                          )
FAC       REPT      (
                           ((=N= V) (=ITER= 1 V (=TIMS= =N= 1)))
                          )
TIM       REPT      (
                           (((U) V) (=ITER= 1 V (=TIMS= 1 U)))
                           (((U UUU) V) (PLU (=REPT= ((U) V)) (0 (*REPT* ((UUU) V)))))
                          )
MIN       REPT      (
                           ((U) (=ITER= 1 U (=MINS= 0 1)))
                           (((U UUU) (V VVV)) ((=MINS= U V) (*REPT* ((UUU) (VVV)))))
                           ((() ()) ())
                           ((U ()) U)
                           ((() U) (=REPT= U))
                          )
```

```
      DIV       REPT    (
                         ((U =N=) (=ITER= I U (=DIVD= I =N=)))
                        )
      POW       REPT    (
                         ((U 1) U)
                         ((U V) (TIM U (=REPT= (U (=DECR= V)))))
                        )
      VAL       REPT    (((=N= U) (=SKEL= A EXPR =N= (=REPT= U *3 (
                         ((U) U)
                         ((U UUU) (=PLUS= U (=TIMS= A (=REPT= (UUU)))))
                        )))))
      PRD       REPT    (((U) (=REPT= U *4 (
                         ((U) U)
                         ((U UUU) (U PL X TI (=REPT= (UUU))))
                        ))))
      PWR       REPT    (((U) (=REPT= (0 U) *5 (
                         ((=N= (0 UUU)) (=REPT= ((=INCR= =N=) (UUU))))
                        ((0 (U UUU)) (U (*WHEN* (UUU) () ()
                                        (PL (*REPT* (1 (UUU)))))))
                        ((1 (U UUU)) ((*WHEN* U 1 () (U TI)) X (*WHEN*
                                (UUU) () () (PL (*REPT* (2 (UUU))))))
                        ((=N= (U)) ((*WHEN* U 1 () (U TI)) X PO =N=))
                        ((=N= (U UUU)) ((*WHEN* U 1 () (U TI)) X PO =N=
                                PL (*REPT* ((=INCR= =N=) (UUU)))))
                        ))))
      ))
(QUOTE (
      U V (UUU) (VVV)
      ))
(LIST)
(QUOTE (*0 (
(==    (=PROG= (WS TS)
       1
       (=PRNT= ==)
       (=REPT= =READ= *1 (
              (E      (=SETQ= WS (VAL =READ= WS)))
              (I       (=SETQ= WS =READ=))
              (N       (=SETQ= WS (NOR WS)))
              (O       (=PRNT= WS))
              (S (=RETN= (THE GRASS IS GREENER)))
              (X (=SKEL= * SKEL =READ= (=PRNT= *)))
                   (Y      (=SETQ= WS (PWR WS)))
              (Z       (=SETQ= WS (PRD WS)))
              (TS       (=SETQ= WS TS))
              ((TS)     (=SETQ= TS WS))
              ((U UUU) (=PROG= () (=REPT= U) (=REPT= (UUU))))
              ))
(=GOTO= 1)
))
)))
)))


))
```